

Technische Universität Ilmenau
Fakultät für Mathematik
und Naturwissenschaften
Institut für Mathematik

Postfach 0565
98684 Ilmenau
Germany
Tel.: 03677/692652
Fax: 03677/691241
Telex: 33 84 23 tuil d.

Preprint No. M 15/95

Berechnung von Matrix-Multiplikationen auf dem PC

Werner Neundorf, Thomas Ortlepp

August 1995

[‡]MSC (1991): 65-04, 65F99, 68-04, 68M07, 68Q25

1 Einleitung

Der Aufwand für die Lösung von Problemen der Numerischen Mathematik wird meist durch die Anzahl der arithmetischen Operationen charakterisiert. Fortschritte können einerseits erzielt werden, wenn neue Algorithmen gefunden werden, die eine Reduktion der Komplexität an Operationen ermöglichen. Andererseits stellt sich aber die Frage, ob der Algorithmus mit weniger arithmetischen Operationen, wenn er auf einer konkreten Rechenmaschine implementiert wird, auch wirklich die Laufzeit verkürzt. Komplizierte Verfahren benötigen leider oft einen höheren Verwaltungsaufwand, und können deshalb dann in der Gesamtlaufzeit ungünstiger sein als ein einfacheres Verfahren mit mehr arithmetischen Operationen, aber mit effizienter und überschaubarer Implementierung.

Es steht also das Problem, ob man nicht für klassische Algorithmen die Eigenschaften des Rechnersystems und/oder des Compilers ausnutzen kann, um die Laufzeit zu optimieren. Möglicherweise bringt eine solche Optimierung letztendlich mehr Gewinn als die Implementierung eines neuen Algorithmus.

In dieser Arbeit untersuchen wir die Fragen für das Problem der Matrixmultiplikation auf PC-Technik. Dabei werden wir vorhandene Verfahren an rechner-spezifische Gegebenheiten anpassen und Einsparungen durch die Verringerung der "Hilfsarbeit" anstreben.

2 Besonderheiten des Rechnersystems

Es ist klar, daß diese Problemstellung sehr von dem verwendeten Rechnertyp abhängt. Auf skalaren Rechnern wird dies zu anderen Ergebnissen führen als auf Hochleistungsrechnern, die Vektorinstruktionen verwenden oder sogar parallele Verarbeitung ermöglichen.

Selbst auf skalaren Rechnern wird die Problemstellung von einer Vielzahl von Parametern beeinflusst, wie etwa :

- schnelle und große Rechenregister, Cache-Speicher,
- interne Parallelität des Rechenwerkes,
- Optimierer, die bestimmte Sprachkonstrukte sehr effektiv behandeln, andere aber nur weniger effektiv,
- Verwendung von Maschinencode.

Auch ohne die explizite Verwendung von Maschinspracheprogrammen ist für die Detailoptimierung ein weites Betätigungsfeld gegeben.

Häufig wird die Rechenzeit in direktem Zusammenhang mit der Anzahl der arithmetischen Operationen gebracht. Das ist ein wichtiges Kriterium. Aber im Zuge der Hardwareentwicklung sollte man andere beachtliche Ergebnisse keinesfalls unberücksichtigt lassen.

So ist zum z.B. eine Multiplikation beim i-Pentium genauso schnell wie eine Addition. Das Quadrieren einer Zahl ist gar doppelt so schnell wie die Addition. Zahlreiche

verbesserte numerische Verfahren beruhen darauf, auf Kosten von Additionen einige Multiplikationen einzusparen (vgl. [1]). In [4] ist der Zeitgewinn untersucht, den diese Algorithmen erbringen. Dabei werden gewisse unrealistische Vorstellungen über den erzielbaren Gewinn häufig korrigiert. Einsparungen an Multiplikationen sind nicht mehr unbedingt sinnvoll, wenn eine Zuweisung mehr Zeit als eine Gleitkommamultiplikation benötigt (beim i-Pentium gar mehr als 200% der Zeit).

Die Kontrollstrukturen verschiedener Compiler für Schleifen, Vergleiche u.v.m. sind sehr unterschiedlich gestaltet. Hier kann man zahlreiche Variationen der Implementation austesten.

Diese und andere Untersuchungen sollte man jedem Optimierungsangriff auf ein numerisches Verfahren vorausschicken. Die speziellen Systemvoraussetzungen sind zunächst zu prüfen. Anschließend kann man den Hebel an den Schwachstellen ansetzen.

2.1 Der Systemtest

Alle Tests wurden auf PC unter dem Betriebssystem MS-DOS gemacht. Zahlreiche Versuche haben gezeigt, daß andere Betriebssysteme derzeit diese Geschwindigkeiten nicht erreichen.

Die Rechenzeit eines Programmtails läßt sich in seine elementaren Bestandteile zerlegen. Eine Bestimmung der einzelnen Geschwindigkeiten gibt oft schon einen Einblick in das Gesamtverhalten. Man kann also schon theoretisch Vorhersagen für die reale Rechenzeit machen. Dabei können jedoch auch so manche Überraschungen auftreten. Das endgültige Verständnis für viele Effekte läßt sich meist erst auf Assemblerebene klären. Hier kann man die Qualität eines Compilers genau erkennen.

Operation	AMD 486DX4-100	i-Pentium 90
Leere Schleife	0.702	0.880
Zuweisung	1.780	1.218
Addition	1.220	0.568
Subtraktion	1.221	0.568
Multiplikation	1.526	0.568
Division	7.416	4.558
Quadrat	1.220	0.232
Wurzel	8.218	7.888

Tab.1. Geschwindigkeiten für 10 Millionen skalare Gleitpunkt-Operationen in *sec*, Borland C++ 3.1 .

Mittels dieser Werte läßt sich nun beispielsweise die Zeit für folgende Kompaktanweisung bestimmen.

for (*i* = 1; *i* < 10000000; *i*++) *c* = *a* * *b*;

Die gemessene Gesamtzeit für den 486DX-PC beträgt 4.01sec. Sie berechnet sich gemäß

$$t = < \text{LeereSchleife} > + < \text{Zuweisung} > + < \text{Multiplikation} >$$

$$= 0.702 + 1.780 + 1.526 = 4.008.$$

Die angegebenen Zeiten sind jedoch stark vom verwendeten Compiler abhängig. Des weiteren versagt diese Idee bei relativ komplexen Programmen, dort können z.B. die Steueranweisungen einiges durcheinander bringen.

In der Tabelle fällt die relativ hohe Arbeitszeit für eine leere Schleife auf. Wodurch entsteht diese? Das Problem ist nicht das Verwalten der Zählvariable. Der Sprung (JMP), den der Prozessor für jeden Durchlauf ausführt, ist die Lösung. Genaugenommen ist es ein bedingter Sprung, der bei allen Intel-Prozessoren die Befehlswarteschlange löscht. Der i-Pentium hat damit noch größere Probleme, da die parallele Pipeline-Struktur hier eher stört als nützt. Daraus ergibt sich eine Anforderung an die Hardwarehersteller bezüglich schneller Speicher.

3 Matrixmultiplikation

Die klassische $n \times n$ Matrixmultiplikation $C = A * B$ braucht n^3 Multiplikationen und $n^3 - n^2$ Additionen.

Um eine untere Zeitschranke für die Multiplikation von Matrizen zu finden, wurde der klassische Algorithmus in Assembler übertragen und in Vergleich zu verschiedenen Compilern gestellt. Hierbei ist festzustellen, daß man oft mit einfachen, gut durchdachten Compilern diese Strukturen schneller verwirklichen kann, als mit so manchem "hochgeachteten Giganten".

3.1 Klassische Matrixmultiplikation

Die klassische Technik ist in zwei Programmiersprachen vorgestellt.

```
type float=double;
procedure klassmul(var c,a,b : matrix);
var i,j,k : word; s : float;
begin
  for i:=1 to n do
    for j:=1 to n do
      begin
        s:=a[i,1]*b[1,j];
        for k:=2 to n do s:=s+a[i,k]*b[k,j];
        c[i,j]:=s;
      end;
    end;
  end;
```

Abb.1. Klassische Matrixmultiplikation, Borland Pascal 7.0 .

```

void matrix ( )
{
    int i,j,k;
    double s;
    for ( i=0; i<n; i++ )
        for ( j=0; j<n; j++ )
            {
                s=0.0;
                for ( k=0; k<n; k++ )
                    s+=a[i][k]*b[k][j];
                c[i][j]=s;
            }
}

```

Abb.2. Klassische Matrixmultiplikation, C++ 3.1, erstes Matricelement ist a_{00} .

Die Erstbelegung der Hilfsvariablen s mit dem Wert $a_{i,1} * b_{1,j}$ bzw. 0 ist nicht ausschlaggebend für die Untersuchungen.

Zusätzliche Zeitverluste sind vorprogrammiert, wenn ohne die Hilfsvariable s gearbeitet wird, d.h. in der inneren Schleife steht

$$\begin{aligned}
 c_{ij} &= c_{ij} + a_{ik} * b_{kj}, \\
 \text{bzw. } c_{ij} &= a_{ik} * b_{kj}.
 \end{aligned}$$

3.2 Aufrolltechnik

Der Grund für die Modifikation durch Aufrollen der Schleifen ist die zeitaufwendige Schleifenverwaltung in den gängigen höheren Programmiersprachen. Die innere Schleifenanweisung k mit ihrer Zuweisung wird nicht mehr n -mal ausgeführt, sondern nur s -mal.

Das basiert auf der Zerlegung $n = ms + r$, $0 \leq r < m$, wobei für Testzwecke $r = 0$ sei. Die Größe m heißt **Abrollparameter** und muß natürlich einen konkreten Wert haben, z.B. $m = 2, 4, 8, \dots$

```

type float=double;
procedure modif_m(var c,a,b : matrix);
var i,j,k : word;
    s : float;
begin
    for i:=1 to n do
        for j:=1 to n do
            begin
                if r=0 then s:=0.0
                else s:=a[i,1]*b[1,j]+a[i,2]*b[2,j]+...+a[i,r]*b[r,j];

```

```

      k:=r+1;
      while k<=n do
        begin
          s:=s+a[i,k]*b[k,j]+a[i,k+1]*b[k+1,j]+...
            +a[i,k+m-1]*b[k+m-1,j];

          k:=k+m
        end;
      c[i,j]:=s;
    end;
end;

```

Abb.3. Aufrolltechnik, Borland Pascal 7.0 .

Wenn $m = 1$ ist, haben wir die klassische Variante.

In [1] sind Ergebnisse dieser Technik an leistungsfhigen Rechnern (Workstation, Großrechner) mit den Sprachen FORTRAN und C sowie der Matrixdimension $n=100, 200, 300$, dargestellt und wie folgt bewertet :

- der Zeitgewinn steigt mit wachsendem Abrollparameter m ,
- wenn $m \approx 15\%$ des Wertes von n erreicht stagniert der Zeitgewinn,
- Compilierung mit Optimierung ist gnstiger,
- der jeweils mgliche maximale Zeitgewinn liegt zwischen 18-70%.

3.3 Blockstrategien

Die Standardversion sei

```

for ( i=0; i<n; i++ )
  for ( j=0; j<n; j++ )
  {
    c[i][j]=0.0;
    for ( k=0; k<n; k++ )
      c[i][j]+=a[i][k]*b[k][j];
  }

```

Abb.4. Einfache Matrixmultiplikation, C++ 3.1 .

Eine erste Verbesserung ist durch die Benutzung einer Hilfsvariablen in der innersten Schleife zu erreichen (siehe Abb.2). Eine weitere Steigerung der Effizienz (speed up) erhalten wir durch die Einfhrung von Zwischenvektoren.

Die folgende Modifikation ist sprachabhngig, sind doch in FORTRAN die Matrizen spaltenweise im Speicher abgelegt, whrend C und PASCAL diese zeilenweise speichern.

Wir betrachten wir das Matrixprodukt $C = A * B$ in FORTRAN. Eine Matrixspalte von B liegt also wie ein Vektor komponentenweise hintereinander im Speicher. So ist es gnstig, diese Spalte in einen Hilfsvektor der Lnge n zu kopieren und diesen dann

n -mal für die Multiplikation mit den n Zeilen von A zu verwenden. Danach wird er durch die nächste Spalte von B neu belegt. Das erfordert in der äußeren Schleifensteuerung i, j einen Tausch der Laufvariablen.

Weitere Versionen nutzen mehrere Zwischenvektoren, wobei die Blockstruktur und der Abrollparameter m aufeinander abgestimmt sind.

In [2] wurde eine Benchmark-Test für die Matrixmultiplikation mit $n=384$ ausgeführt. Erreichbar war maximal eine Leistung von 66 MFlops. Die verschiedenen Algorithmen liegen weit entfernt von diesem Bestwert. Auch speziell auf die Dimension und Struktur zugeschnittene Versionen können noch nicht ganz befriedigen.

Lfd. Nr.	Kode-Type/Optimierungsstrategie	MFlops
1	Einfache Matrixmultiplikation	2.1
2	mit Hilfsvariable für inneres Skalarprodukt	2.4
3	mit Zwischenvektor für Umspeicherung der Spalten der Matrix B	9.3
4	mit 2 Zwischenvektoren für Umspeicherung von 2 aufeinanderfolgenden Spalten sowie der Summation von 2x2-Matrizen durch Aufrollen, $m=2$	17.6
5	mit 4 Zwischenvektoren für Umspeicherung von 4 aufeinanderfolgenden Spalten sowie der Summation von 4x4-Matrizen durch Aufrollen, $m=4$	24.5
6	Spezieller 2-Stufen-Block-Algorithmus, äußere Blockstruktur: 96x96, innere Blockstruktur: 3x4, Aufrolltechnik, $m=4$	34.6

Tab.2. Leistung in MFlops ($=10^6$ Operationen pro *sec*) verschiedener Techniken der Matrixmultiplikation auf HP9000/750 mit FORTRAN.

4 Matrixmultiplikation am PC

Die Anzahl der Multiplikationen und Additionen ist ohne grundsätzliche Änderungen am Verfahren nicht zu verringern.

Ziel bleibt die Optimierung der Matrixmultiplikation durch die Verkürzung der innersten Schleife. Dabei kann die Zahl der Zuweisungen für die Hilfsvariable s verkleinert werden. Das bringt, wie Tabelle 1 zeigt, bereits einen meßbaren Effekt. Es kommt also lediglich bei den Kontrollstrukturen zu einer Einsparung.

Die nachfolgenden Programmteile arbeiten ohne Zwischenvektoren und sind in C++ angegeben. Der Komponententyp der Matrizen ist *double* (8 Byte).

Der klassische Algorithmus hat die folgende Gestalt.

```
for ( i=0; i<n; i++ )
  for ( j=0; j<n; j++ )
  {
    s=0.0;
    for ( k=0; k<n; k++ )
      s+=a[i][k]*b[k][j];
    c[i][j]=s;
  }
```

Abb.5. Programm 1, Version STD.

Die erste Verbesserung durch Aufrollen der Schleifen hat die allgemeine Form.

```
for ( i=0; i<n; i++ )
  for ( j=0; j<n; j++ )
  {
    s=a[i][0]*b[0][j];
    for ( k=1; k<(n % m)-1; k++ )    // % Modulo
      s+=a[i][k]*b[k][j];
    for ( k=(n % m); k<n; k+=m )
      s+=a[i][k]*b[k][j]+...+a[i][k+m-1]*b[k+m-1][j];
    c[i][j]=s;
  }
```

Abb.6. Programm mit Aufrolltechnik, m Abrollparameter.

Unter der Annahme der Teilbarkeit der Matrixdimension n durch den Abrollparameter m entsteht eine etwas einfachere Darstellung.

```
for ( i=0; i<n; i++ )
  for ( j=0; j<n; j++ )
  {
    s=0.0;
    for ( k=0; k<n; k+=m )
      s+=a[i][k]*b[k][j]+...+a[i][k+m-1]*b[k+m-1][j];
    c[i][j]=s;
  }
```

Abb.7. Programm 2 mit Aufrolltechnik, Version $M=m$.

Im konkreten Programm muß man sich für ein festes m entscheiden. Man kann dieses Aufrollen jedoch auch durch einen einfachen Prozessor erledigen lassen.

In den bisherigen Programmen treten ständig indizierte Adressierungen auf. Wer einmal versucht hat, diese in Assemblersprache zu übertragen, wird feststellen, daß sie

einen erheblichen Aufwand enthalten. Aus beiden Indizes muß laufend eine physikalische Adresse gebildet werden.

Man kann dieses Verfahren vereinfacht so beschreiben:

$$ADR(a[i][j]) = ADR(a[0][0]) + i * sizeof(a[0]) + j * sizeof(a[0][0]),$$

wobei gilt

- $ADR(a[i][j])$ Adresse des Matrixelements $a[i][j]$,
- $ADR(a[0][0])$ Adresse des ersten Matrixelements,
- $sizeof(a[0][0])$ Größe eines Elements in Bytes,
- $sizeof(a[0])$ Größe einer Zeile/Spalte in Bytes.

Es ist zu erkennen, daß zwei Additionen und zwei Multiplikationen im 32-Bit-Integer-Bereich auftreten. Für einen wahlfreien Zugriff ist diese Methode auch unvermeidbar. In diesem speziellen Fall ist jedoch zu bemerken, daß sich die Berechnung vereinfachen läßt. Für ein benachbartes Element (horizontal oder vertikal) muß die Adresse jeweils um einen festen Wert erhöht werden. Damit kann man dem Prozessor die 1stige Adreßberechnung abnehmen.

Die Standardversion mit Adreßmanipulation lautet.

```
d=sizeof(a[0])/sizeof(a[0][0]); // Laenge einer Zeile/Spalte
                                // Zeilengroesse in Elementen, =n
e=d/m;                          // Laenge der innersten Schleife
for ( i=0; i<n; i++ )
  for ( j=0; j<n; j++ )
  {
    x=&a[i][0];                  // Adressbestimmung des Elements
    y=&b[0][j];
    s=0.0;
    for ( k=0; k<n; k++ )
    {
      s+=(*x++)*(y);
      y+=d;
    }
    c[i][j]=s;
  }
```

Abb.8. Programm 3, Version ADR,

a : Adreßerhöhung um 1 bei $a_{ik} \Rightarrow a_{i,k+1}$,

b : Adreßerhöhung um d bei $b_{kj} \Rightarrow b_{k+1,j}$.

Die Übersetzung dieser Konstruktion weist bei den verschiedenen Compilern große Unterschiede auf.

Die Verknüpfung der Varianten ADR mit $M=m$ (Programme 2,3) liefert die Version ADR $M=m$.

```

m=2;
d=sizeof(a[0])/sizeof(a[0][0]); // Laenge einer Zeile/Spalte
                                // Zeilengroesse in Elementen, =n
e=d/m;                          // Laenge der innersten Schleife
for ( i=0; i<n; i++ )
    for ( j=0; j<n; j++ )
        { x=&a[i][0];           // Adressbestimmung des Elements
          y=&b[0][j];
          s=0.0;
          for ( k=0; k<e; k++ )
              { s+=(*x++)*(y);
                y+=d;
                s+=(*x++)*(y);
                y+=d;
              }
          c[i][j]=s;
        }

```

Abb.9. Programm 4, Version ADR $M=m$, $m=2$.

Für relativ kleine Matrizen sind die Vorteile der Modifikation beschränkt. Der Wert für die hier verwendete Matrixdimension $n = 48$ ist das relative Maximum an Zeitgewinn. Für großdimensionale Probleme kann kein weiterer relativer Zeitgewinn verzeichnet werden.

Verfahren	TC 1.0	BC 3.1	BC 4.5	TP 6.0	TP 7.0	BP 7.0
STD	26.852	25.211	26.750	29.00	27.57	30.37
M=2	23.680	21.309	22.520	25.42	26.64	26.86
M=4	23.008	20.492	21.469	24.06	25.01	25.37
M=8	22.680	19.609	20.441	24.03	25.00	25.35
ADR	14.609	21.199	20.160	22.80	26.68	24.55
ADR M=2	15.987	19.109	21.031	22.34	25.60	23.89
ADR M=4	15.875	18.781	21.198	21.89	24.98	23.51
ADR M=8	15.883	18.570	21.091	21.48	24.83	23.23
% von ASM	162	206	224	238	276	258

Tab.3. Rechenzeit für verschiedene Compiler (in *sec*) auf PC AMD 486DX4-100 bei 200 Matrixmultiplikationen $C = A * B$ mit $n = 48$.

Für die letzte Zeile der Tabelle 3 wurde der klassische Algorithmus, mit innerer Schleife in Assembler programmiert, als Vergleichsbasis gewählt. Die Version ASM benötigt **9.01sec**. Das angegebene Verhältnis in % bezieht sich jeweils auf den besten Wert für den entsprechenden Compiler.

```

procedure mul5(var c,a,b : matrix);
var i,j,k : word; s : float;
begin
  for i:=1 to n do
    for j:=1 to n do
      begin
        asm
          MOV     BX,flong
          MOV     AX,i
          DEC     AX
          MUL     BX
          MOV     CX,AX
          MOV     AX,dm
          MUL     BX
          MOV     BX,AX
          MOV     DX,j
          DEC     DX
          MUL     DX
          MOV     DX,AX
          MOV     AX,n
          DEC     AX
          PUSH    DS
          LES     DI,a
          ADD     DI,DX
          LDS     SI,b
          ADD     SI,CX
          MOV     CX,AX
          FINIT
          FLD     qword ptr ES:[DI]
          FMUL    qword ptr DS:[SI]
        @M1:    ADD     SI,BX
          ADD     DI,flong
          FLD     qword ptr ES:[DI]
          FMUL    qword ptr DS:[SI]
          FADDP   ST(1),ST(0)
          LOOP    @M1
          POP     DS
          FSTP    s
        end;
        c[i,j]:=s;
      end
    end
  end;
end;

```

Abb.10. Programm 5, Version ASM.

Das Aufrollen der Schleifen bringt einen Zeitvorteil. Er ist jedoch beschränkt. Die Erkenntnis aus Abschnitt 3.2 bzgl. des Abrollparameters m hat sich auch hier bestätigt, so daß größere Werte wie 12,16,... keine Vorteile mehr bringen.

Mehr verspricht da schon die Anwendung der Adreßrechnung. In PASCAL ist diese jedoch recht umständlich und nur durch einen Trick zu realisieren (Abb. 11).

In C kann man sie aber recht gewinnbringend einsetzen (siehe Abb.9).

```
const dim = 50;
type float = double;
    matrix = array[1..dim,1..dim] of float;
    BF      = array[0..100] of float;

procedure adr1(var c,a,b : matrix );
var i,j,k,d : word;
    s      : float;
    x,y    : pointer;
begin
    d:=dim;
    for i:=1 to n do
        for j:=1 to n do
            begin
                x:=@a[i,1];
                y:=@b[1,j];
                s:=float(x^)*float(y^);
                for k:=2 to n do
                    begin
                        x:=@BF(x^)[1];
                        y:=@BF(y^)[d];
                        s:=s+float(x^)*float(y^);
                    end;
                c[i,j]:=s;
            end
        end
    end;
```

Abb.11. Adressierungsmechanismus für Felder in PASCAL.

Die Werte für die erste und dritte Spalte der Adreßversion in der Tabelle 3 lassen eine Frage aufkommen. Wie kann es sein, daß das Aufrollen der Schleife die Rechenzeit verschlechtert? Hier lassen sich mit dem Debugger zwei unterschiedliche Gründe offenbaren.

In Turbo C++1.0 werden Schleifen mit einer Codegröße von nicht mehr als 256 Bytes anders implementiert. Hier ist also ein kleiner Schleifenkörper optimaler.

In Borland C++4.5 liegt es jedoch an der immensen Größe des Codes. Damit wird der rechner-spezifische Cache-Speicher entscheidend. Die erste Variante ADR M=2

lßt sich noch im 8KBytes großen First Level Cache unterbringen. Der Compiler beachtet dies auch. Die anderen aufgerollten Schleifen berschreiten jedoch diese GröÙe und werden somit etwas langsamer. An diesem Fall ist wieder einmal deutlich zu sehen, wie die verwendete Hardware mit Softwarekonstrukten zusammenhngt.

4.1 Aufrolltechnik im Assembler

Die wohl allem vorzuziehende Variante ist somit eine Implementierung der Aufrolltechnik in Assemblersprache.

Unter denselben Voraussetzungen wie in Tabelle 3 wurde ein Vergleich mit der Standardversion auf zwei PC-Typen vorgenommen.

Verfahren	AMD 486DX4-100	i-Pentium 90
STD	9.01	4.39
M=2	8.42	4.01
M=4	7.91	3.68
M=8	7.82	3.67

Tab.4. Rechenzeit bei Aufrolltechnik fr PASCAL mit ASM (in *sec*) auf PC.

5 Schlußbemerkung

Wir erkennen, daß die Programmierung gewisser Strukturen in Assemblersprache durchaus von Nutzen sein kann. Es ist aber nicht zu erwarten, daß jeder nun mhsam Assembler lernt, um sein ganz spezielles Problem bis zu letzten Millisekunde zu optimieren. Es geht auch um das Verstdnis der interen Ablufe in einem Computer. Dies stellt ein wichtiges Hilfsmittel fr eine effektive Programmierung dar.

Natrlich lassen selbst die angebotenen Compiler bei der gesehenen Streuung der Mglichkeiten und Geschwindigkeiten auf einige nicht unerhebliche Reserven hoffen.

Literatur

- [1] SLAVKOVSKY, P.; RDE, U.: *Schnellere Berechnung klassischer Matrix-Multiplikationen*. Preprint TUM-I9032, SFB-Bereich Nr. 342/17/90 A, Mnchen September 1990.
- [2] BONK, T.; RDE, U.: *Perfomance Analysis and Optimization of Numerically Intensive Programs*. Preprint TUM-I9238, SFB-Bereich Nr. 342/26/92 A, Mnchen November 1992.
- [3] STOER, J.: *Einfhrung in die Numerische Mathematik. Band 1*. Springer-Verlag Berlin 1979.
- [4] SPIESS, J.: *Untersuchungen des Zeitgewinns durch neue Algorithmen zur Matrix-Multiplikationen*. Computing 17, 23-36 (1976).
- [5] ORTLEPP, TH.: *Schnellere Berechnung der klassischen Matrix-Multiplikation auf PC*. Beleg IfMath TUIlmenau 1995.
- [6] KIELBASINSKI, A.; SCHWETLICK, H.: *Numerische lineare Algebra*. Mathematik fr Naturwissenschaft und Technik Band 18, DVW, Berlin 1988.

Anschrift:

Dr. Werner Neundorf, stud. cand. Thomas Ortlepp
Institut fr Mathematik
Technische Universitt Ilmenau
PF 0565
D - 98684 Ilmenau

e-mail : neundorf@mathematik.tu-ilmenau.de